Storage and Database Management for Big Data

Vijay Gadepally, Jeremy Kepner and Albert Reuther

MIT Lincoln Laboratory

Lexington, MA, USA 02420

vijayg@ll.mit.edu, kepner@ll.mit.edu, reuther@ll.mit.edu

Distribution A: Public Release

July 27, 2015

Contents

Chapter 1

Storage and Database Management for Big Data

1.1   Introduction

The ability to collect and analyze large amounts of data is a growing problem within the scientific community. The growing gap between data and users calls for innovative tools that address the challenges faced by big data volume, velocity and variety. While there has been great progress in the world of database technologies in the past few years, there are still many fundamental considerations that must be made by scientists. For example, which of the seemingly infinite technologies are the best to use for my problem? Answers to such questions require a careful understanding of the technology field in addition to the types of problems that are being solved. This chapter aims to address many of the pressing questions faced by individuals interested in using storage or database technologies to solve their big data problems.

Storage and database management is a vast field with many decades of results from very talented scientists and researchers. There are numerous books, courses and articles dedicated to the study. This chapter attempts to highlight some of these developments as they relate to the equally vast field of big data. However, it would be unfair to say that this chapter provides a comprehensive analysis of the field - such a study would require many volumes. It is our hope that this chapter can be used as a launching pad for researchers interested in the study. Where possible, we highlight important studies that can be pursued for further reading.

In Section 1.2, we discuss the big data challenge as it relates to storage and database engines. The chapter goes on to discuss database utility compared to large parallel storage arrays. Then, the chapter discusses the history of database management systems with special emphasis on current and upcoming database technology trends. In order to provide readers with a deeper understanding of these technologies, the chapter will provides a deep dive into two canonical open source database technologies: Apache Accumulo, [1] which is based on the popular Google BigTable design, and a NewSQL array database called SciDB [59]. Finally, we will provide insight into technology selection and walk readers through a case study which highlights the use of various database technologies to solve a medical big data problem.

1.2   The Big Data Challenge

Working with Big Data is prone to a variety of challenges. Very often, these challenges are referred to as the three Vs of Big Data: Volume, Velocity and Variety [45]. Most recently, there has been a new emergent challenge (perhaps a fourth V): Veracity. These combined challenges constitute a large reason why Big Data is so difficult to work with.

Big data volume stresses the storage, memory, and computational capacity of a computing system and often requires access to a computing cloud. The National Institute of Science and Technology (NIST) defines cloud computing to be "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources ... that can be rapidly provisioned and released with minimal management effort or service provider interaction" [47]. Within this definition, there are different cloud models that satisfy different problem
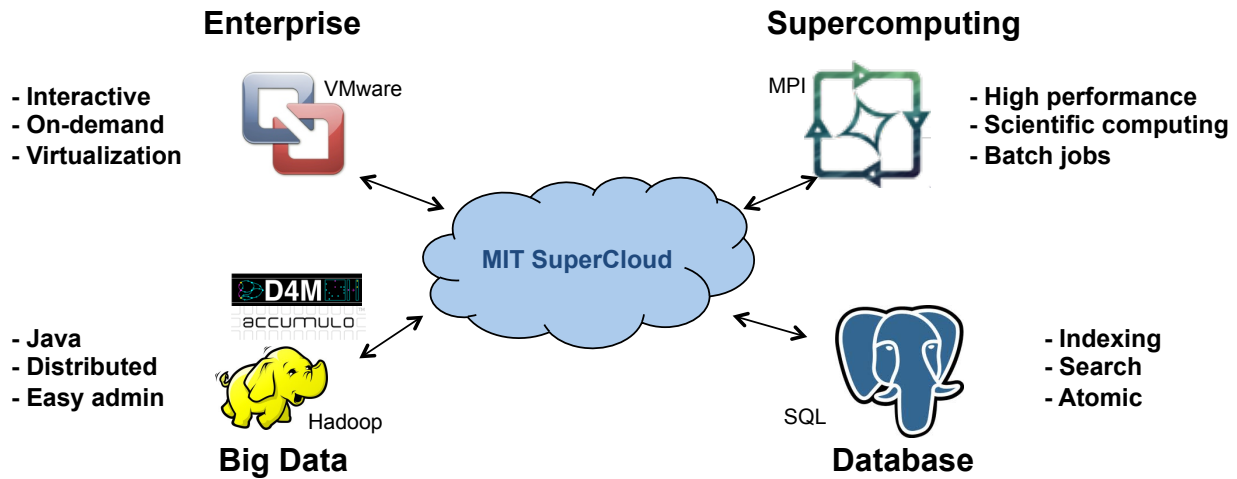
Figure 1.1: The MIT SuperCloud infrastructure allows multiple cloud environments to be launched on the same hardware and software platform in order to address big data volume.

characteristics and choosing the right cloud model is problem specific. Currently, there are four multi-billion dollar ecosystems that dominate the cloud-computing landscape: enterprise clouds, big data clouds, SQL database clouds, and supercomputing clouds. Each cloud ecosystem has its own hardware, software, conferences, and business markets. The broad nature of business big data challenges make it unlikely that one cloud ecosystem can meet its needs, and solutions are likely to require the tools and techniques from more than one cloud ecosystem. For this reason, at the Massachusetts Institute of Technology (MIT) Lincoln Laboratory, we developed the MIT Super-Cloud architecture [51] that enables the prototyping of four common computing ecosystems on a shared hardware platform as depicted in Figure 1.1. The velocity of big data stresses the rate at which data can be absorbed and meaningful answers produced. Very often, the velocity challenge is mitigated through high performance databases, file systems and/or processing. Big data variety may present the largest challenge and greatest opportunities. The promise of big data is the ability to correlate diverse and heterogeneous data to form new insights. A new fourth V [26], veracity challenges our ability to perform computation on data while preserving privacy.

As a simple example of the scale of data and how it has changed in the recent past, consider the social media analysis developed by [24]. In 2011, Facebook had approximately 700,000 pieces of content per minute; Twitter had approximately 100,000 tweets per minute; and YouTube had approximately 48 hours of video per minute. By 2015, just four years later, Facebook had 2.5 million pieces of content per minute; Twitter had approximately 277,000 tweets per minute; and

YouTube had approximately 72 hours of new video per minute. This increase in data generated can be roughly approximated to be 350 MB/min for Facebook, 50MB/min for Twitter, and 24 to 48 GB/min for YouTube! In terms of the sheer volume of data, IDC estimates that from the year 2005 to the year 2020, there will an increase in the amount of data generated from 130 exabytes to 40,000 exabytes [30].

One of the greatest Big Data challenges is in determining the ideal storage engine for a large data set. Databases and filesystems provide access to vast amounts of data but differ at a fundamental level. Filesystem storage engines are designed to provide access to a potentially large subset of the full dataset. Database engines are designed to index and provide access to a smaller, but well defined, subset of data. Before looking at particular storage and database engines, it is important to take a look at where these systems fall within the larger big data system.

1.3   System Engineering for Big Data

Systems engineering studies the development of complex systems. Given the many challenges of Big Data as described in Section 1.2, systems engineering has a great deal of applicability to developing a Big Data system. Once convenient way to visualize a Big Data system is as a pipeline. In fact, most Big Data systems consist of different steps which are connected to each other to form a pipeline (sometimes, they may not be explicitly separated though that is the function they are performing). Figure 1.2 shows a notional pipeline for Big Data processing.

First, raw data is often collected from sensors or other such sources. These raw files often come in a variety of formats such as comma separated values (CSV), JavaScript Object Notation (JSON) [21], or other proprietary sensor formats. Most often, this raw data is collected by the system and placed into files that replicate the formatting of the original sensor. Retrieval of raw data may be done by different interfaces such as cURL (http://curl.haxx.se/) or other messaging paradigms such as publish/subscribe. The aforementioned formats and retrieval interfaces are by no means exhaustive but highlight some of the popular tools being used.

Once the raw data is on the target system, the next step in the pipeline is to parse these files
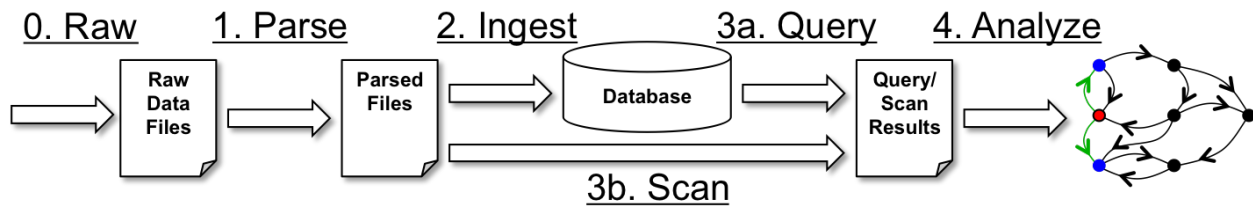
Figure 1.2: A standard big data pipeline consists of five steps to go from raw data to useful analytics.

into a more readable format or to remove components that are not required for the end-analytic. Often, this step involves removing remnants of the original data collection step such as unique identifiers that are no longer needed for further processing. The parsed files are often kept on a serial or parallel file system and can be used directly for analytics by scanning files. For example, a simple word count analytic can be done by using the Linux grep command on the parsed files, or more complex analytics can be performed by using a parallel processing framework such as Hadoop MapReduce or the Message Passing Interface (MPI). As an example of an analytic which works best directly with the file system, dimensional analysis [27] performs aggregate statistics on the full dataset and is much more efficient working directrly from a high performance parallel file system.

For other analytics (especially those that wish to access only a small portion of the entire dataset), it is convenient to ingest this data into a suitable database. An example of such an analytic is given in [28] which performs an analysis on the popularlity of particular entities in a database. This example takes only a small, random piece of the dataset (the counts of words is much smaller than the full dataset) and is well suited for database usage. Once data is in the database or on the filesystem, a user can write queries or scans depending on their use case to produce results that can then be used for complex analytics such as topic modeling.

Each step of the pipeline involves a variety of choices and decisions. These choices may depend on hardware, software or other factors. Many of these choices will also make a difference to the later parts of the pipeline and it is important to make informed decisions. Some of the choices that one may have at each step:

- Step 0: Size of individual raw data files, output format

- Step 1: Parsed data contents, data representation, parser design

- Step 2: Size of database, number of parallel processors, pre-processing

- Step 3: Scan or query for data, use of parallel processing

- Step 4: Visualization tools, algorithms

For the remainder of this chapter, we will focus on some of the decisions in steps two and three of the pipeline. By the end of the chapter, we hope that readers will have an understanding of different storage and database engines, the right time to use technology, and how these pieces can come together.

1.4   Disks and Filesystems

One of the most common ways to store a large quantity of data is through the use of traditional storage media such as hard drives. There are many storage options that must be carefully considered that depend upon various parameters such as total data volume and desired read and write rates. In the pipeline of Figure 1.2, the storage engine plays an important part of steps two and three.

In order to deal with many challenges such as preserving data through failures, the past decades have seen the development of many technologies such as RAID (redundant array of independent disks) [17], NFS (network file system), HDFS (Hadoop Distributed File System) [11], and Lustre [67]. These technologies aim to abstract the physical hardware away from application developers in order to provide an interface for an operating system to keep track of a large number of files while allowing support for data failure, high speed seeks, and fast writes. In this section, we will focus on two leading technologies, Lustre and HDFS.

1.4.1 Serial Memory and Storage

The most prevalent form of data storage is provided by an individual's laptop or desktop system. Within these systems, there are different levels of memory and storage that trade off speed with cost calculated as bytes per dollar. The fastest memory provided by a system (apart from the relatively low capacity system cache) is the main memory or random access memory (RAM). This volatile memory provides relatively high speed (10s of GB/s in 2015) and is often used to store data up to hundreds of gigabytes in 2015. When the data size is larger than the main memory, other forms of storage are used. Within serial storage technologies, some of the most common are traditional spinning magnetic disc hard drives and solid state drives (solid state drives may be designed to use volatile RAM or non-volatile flash technology). The capacity of these technologies can be in the 10s of TB each and can support transfer rates anywhere from approximately 100MB/s to GB/s in 2015.

1.4.2 Parallel Storage: Lustre

Lustre is designed to meet the highest bandwidth file requirements on the largest systems in the world [12] and is used for a variety of scientific workloads [49]. The open source Lustre parallel file system presents itself as a standard POSIX, general-purpose file system and is mounted by client computers running the Lustre client software. Files stored in Lustre contain two components metadata and object data. Metadata consists of the fields associated with each file such as i-node, filename, file permissions, and timestamps. Object data consists of the binary data stored in the file. File metadata is stored in the Lustre metadata server (MDS). Object data is stored in object storage servers (OSSes) shown in Figure 1.3. When a client requests data from a file, it first contacts the MDS which returns pointers to the appropriate objects in the OSSes. This movement of information is transparent to the user and handled fully by the Lustre client. To an application, Lustre operations appear as standard file system operations and require no modification of application code.

A typical Lustre installation might have many OSSes. In turn, each OSS can have a large number of drives that are often formatted in a RAID configuration (often RAID6) to allow for
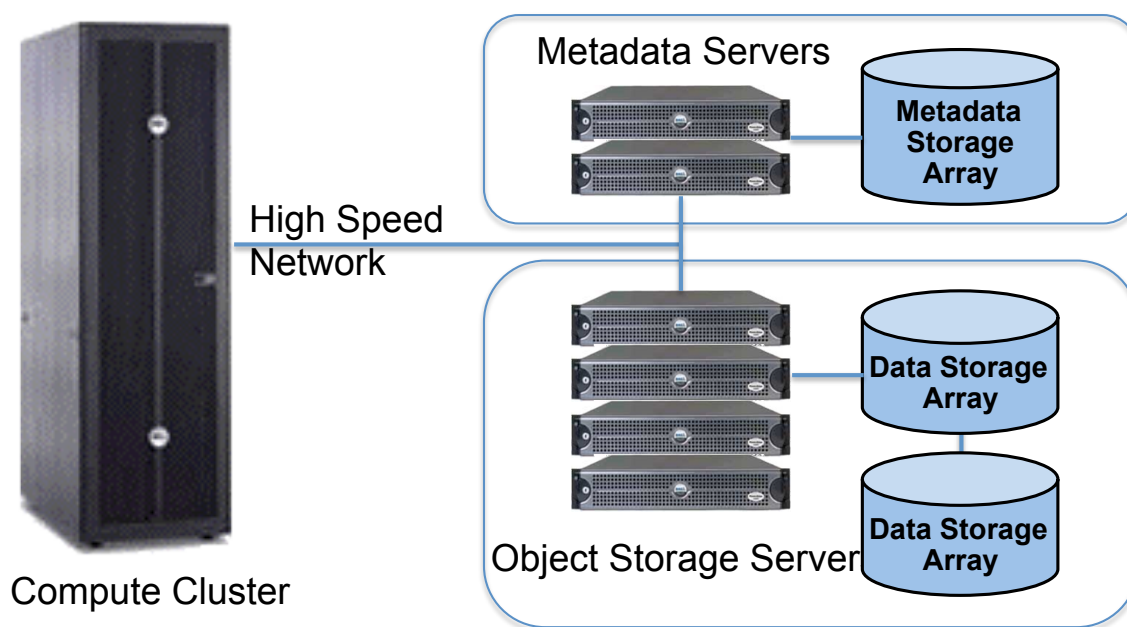
Figure 1.3: A Lustre installation consists of metadata servers and object storage servers. These are connected to a compute cluster via a high speed interconnect such at 10Gb Ethernet or Infiniband.

the failure of any two drives in an OSS. The many drives in an OSS allows data to be read in parallel at high bandwidth. File objects are striped across multiple OSSes to further increase parallel performance. The above redundancy is designed to give Lustre high availability while avoiding a single point of failure. Data loss can only occur if three drives fail in the same OSS prior to any one of the failures being corrected. For Lustre, the typical storage penalty to provide this redundancy is approximately 35%. Thus, a system with 6 petabytes of raw storage will provide 4 petabytes of data capacity to its users.

Lustre is designed to deliver high read and write performance to many simultaneous large files. Lustre systems offer very high bandwidth access to data. For a typical Lustre configuration, this bandwidth may be approximately 12 GB/second in 2015 [2]. This is achieved by the clients having a direct connection to the OSSes via a well-designed high speed network. This connection is brokered by the MDS. The peak bandwidth of Lustre is determined by the aggregate network bandwidth to the client systems, the bisection bandwidth of the network switch, the aggregate network connection to the OSSes, and the aggregate bandwidth of the all the disks [42]. Like most file systems, Lustre is designed for sequential read access and not random lookups of data (unlike a database). To find a particular data value in Lustre requires, on average, scanning through half
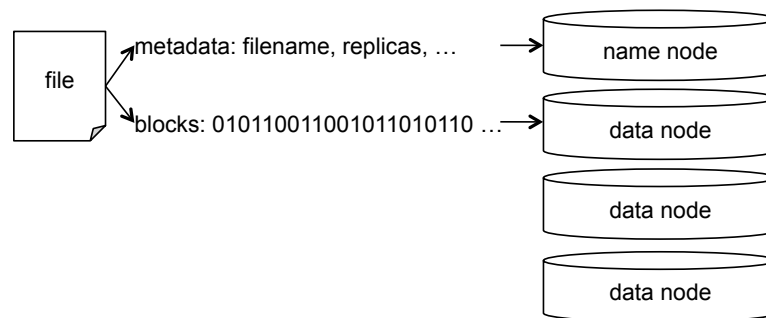
Figure 1.4: Hadoop Splits a file into metadata and replicates it in data blocks.

the file system. For a typical system with approximately 12 GB/second of maximum bandwidth and four petabytes of user storage, this may require approximately four days.

1.4.3 Parallel Storage: Hadoop Distributed File System (HDFS)

Hadoop is a fault-tolerant, distributed file system and distributed computation system. An important component of the Hadoop ecosystem is the supporting filesystem called the Hadoop Distributed File System (HDFS) that enables MapReduce [22] style jobs. HDFS is modeled after the Google File System (GFS) [33] and is a scalable distributed file system for large, distributed, and data-intensive applications. GFS and HDFS provide fault tolerance while running on inexpensive off-the-shelf hardware, and deliver high aggregate performance to a large number of clients. The Hadoop distributed computation system uses the map-reduce parallel programming model for distributing computation onto the data nodes.

The foundational assumptions of HDFS are that its hardware and applications have the following properties [11]: high rates of hardware failures, special purpose applications, large data sets, write-once-read-many data, and read-dominated applications. HDFS is designed for an important, but highly specialized class of applications for a specific class of hardware. In HDFS, applications primarily employ a co-design model whereby the HDFS file system is accessed via specific calls associated with the Hadoop API.

A file stored in HDFS is broken into two pieces: metadata and data blocks as shown in Figure 1.4. Similar to the Lustre filesystem, metadata consists of fields such as the filename, creation

date, and the number of replicas of a particular piece of data. Data blocks consist of the binary data stored in the file. File metadata is stored in an HDFS name node. Block data is stored on data nodes. HDFS is designed to store very large files that will be broken up into multiple data blocks. In addition, HDFS is designed to support fault-tolerance in massive distributed data centers. Each block has a specified number of replicas that are distributed across different data nodes. The most common HDFS replication policy is to store three copies of each data block in a location aware manner so that one replica is on a node in the local rack, the second replica on a node in a different rack and the third replica on another node in the same different rack [3]. With such a policy, the data will be protected from node and rack failure.

The storage penalty for a triple replication policy is 66%. Thus, a system with six petabytes of raw storage will provide two petabytes of data capacity to its users with triple replication. Data loss can only occur if three drives fail prior to any one of the failures being corrected. Hadoop is written in Java and is installed in a special Hadoop user account that runs various Hadoop daemon processes to provide services to connecting clients. Hadoop applications contain special API calls to access the HDFS services. A typical Hadoop application using the map-reduce programming model will distribute an application over the file system so that each application is exclusively reading blocks that are local to the node on which it is running. A well-written Hadoop application can achieve very high performance if the blocks of the files are well distributed across the data nodes. Hadoop applications use the same hardware for storage and computation. The bandwidth achieved out of HDFS is highly dependent upon the computation to communication ratio of the Hadoop application. For a well designed Hadoop application, this aggregate bandwidth may be as high as 100 GB/seconds for a typical HDFS setup. Like most other file systems, HDFS is designed for sequential data access and no random access of data.

1.5   Database Management Systems

Relational or SQL (Structured Query Language) databases [20, 62] have been the de facto interface to databases since the 1980s and are the bedrock of electronic transactions around the world. For example, most financial transactions in the world make use of technologies such as Oracle or

dBase. With the great rise in quantity of unstructured data and analytics based on the statistical properties of datasets, NoSQL (Not Only SQL) database stores such as the Google BigTable [19] have been developed. These databases are capable of processing the large heterogenous data collected from the Internet and other sensor platforms. One style of NoSQL databases which have become used for applications that require support for high velocity data ingest and relatively simple cell-level queries are key-value stores.

As a result, the majority of the volume of data on the Internet is now analyzed using key-value stores such as Amazon Dynamo [23], Cassandra [44], and HBase [32]. Key-value stores and other NoSQL databases compromise on data consistency in order to provide higher performance. In response to this challenge, the relational database community has developed a new class of relational databases (often referred to as NewSQL) such as SciDB [16], H-Store [37], VoltDB [64] to provide the features of relational databases while also scaling to very large data sets. Very often, these newSQL databases make use of a different datamodel [16] or advances in hardware architectures. For example, MemSQL [56] is a distributed in-memory database that provides high performance, ACID compliant relational database management. Another example, BlueDBM [36], provides high performance data access through flash storage and field programmable gate arrays (FPGA).

In this section, we provide an overview of database management systems, the different generations of databases, and a deep dive into two newer technologies, a key-value store: Apache Accumulo and an array database - SciDB.

1.5.1 Database Management Systems and Features

A database is a collection of data and all of the supporting data structures. The software interface between users and a database is known as the database management system. Database management systems provide the most visible view into a dataset. There are many popular database management systems such as MySQL [4], PostgreSQL [63], and Oracle [5]. Most commonly, users interact with database management systems for a variety of reasons:

1. To define data, schema, and ontologies

2. To update/modify data in the database

3. To retrieve or query data

4. To perform database administration or modify parameters such as security settings

5. More recently, to perform analytics on the data within the database

Databases are used to support data collection, indexing and retrieval through transactions. A database transaction refers to the collection of steps involved in performing a single task [31]. For example, a single financial transaction such as "credit $100 towards the account of John Doe" may involve a series of steps such as locating the account information for John Doe, determining the current account value, adding $100 to the account, and ensuring that this new value is seen by any other transaction in the future. Different databases provide different guarantees on what happens during a transaction.

Relational databases provide ACID guarantees: atomicity, consistency, isolation and durability. Atomicity provides the guarantee that database transactions either occur fully or completely fail. This property is useful to ensure that parts of a transaction do not occur successfully if other parts fail, which may lead to an unknown state. The second guarantee, consistency, is important to ensure that all parts of the database see the same data. This guarantee is important to ensure that when different clients perform transactions and query the database, they see the same results. For example, in a financial transaction, a bank account may be debited before further transactions can occur. Without consistency, parts of the database may see different amounts of money (not a great database property!). Isolation in a database refers to a mechanism of concurrency control in a database. In many databases, there may be numerous transactions occurring at the same time. Isolation ensures that these transactions are isolated from other concurrent transactions. Finally, database durability is the property that when a transaction has completed, it is persisted even if the database has a system failure. Non-relational databases such as NoSQL databases often provide a relaxed version of ACID guarantees referred to as BASE guarantees in order to support a distributed architecture or performance. This stands for Basically Available, Soft State, Eventual Consistency guarantees [50]. As opposed to the ACID guarantees of relational databases, non relational databases do not provide strict guarantees on the consistency of each transaction but

instead provide a looser guarantee that *eventually* one will have consistency in the database. For many applications, this may be an acceptable guarantee.

For these reasons, financial transactions employ relational databases that have the strong ACID guarantees on transactions. More recent trends that make use of the vast quantity of data retrieval from the Internet can be done via non-relational databases such as Google BigTable [19], which are responsible for fast access to information. For instance, calculating statistics on large datasets are not as susceptible to small eventual changes to the data.

While many aspects of learning how to use a database can be taught through books or guides such as this, there is an artistic aspect to their usage as well. More practice and experience with databases will help overcome common issues, improved performance tuning, and help with improved database management system stability. Prior to using a database, it is important to understand the choices available, properties of the data and key requirements.

1.5.2 History of Open Source Databases and Parallel Processing

Databases and parallel processing have developed together over the past few decades. Parallel processing is the ability to take a given program and split it across multiple processors in order to reduce computation time or resource availability for the application. Very often, advances in parallel processing are directly used for the computational piece of databases such as sorting and indexing datasets.

Open source databases have been around since the mid-1990s. Some of the first relational databases were based on the design of the Ingres database [62] originally developed at UC Berkeley. During the same time period, there were many parallel processing or high performance computing paradigms [53, 38] that were being developed by industry and academia. The first few (popular) open source databases that were created were PostgreSQL and MySQL. The earliest forms of parallel cluster processing take their root in the early-1990s with the wide proliferation of *nix based operating systems and parallel processing schedulers such as Grid Engine. For about ten years, until the mid-2000s, these technologies continued to mature, and developers saw the need for greater adoption of distributed computing and databases.

Based on a series of papers from Google in the mid-2000s, the MapReduce computing paradigm was created which gained wide acceptance through the open source Apache Hadoop soon after. These technologies, combined with the seminal Google BigTable [19] paper helped spark the NoSQL movement in databases. Not long after this, numerous technologies such as GraphLab [46], Neo4j [68], and Giraph [9] were developed to apply parallel processing to large unstructured graphs such as those being collected and stored in NewSQL databases. Since the year 2010, there has been renewed interest in developing technologies that offer high performance along with some of the ACID guarantees of relational databases (which will be discussed in the next section). This requirement has driven the development of a new generation of relational databases often called NewSQL. In the parallel processing world, users are looking for better ways to deal with streaming data or machine learning and graph algorithms than the Hadoop framework offered and are developing new technologies such as Apache Storm [65] and Spark [69]. Of course, the worlds of parallel processing and databases will continue to evolve, and it will be interesting to see what lies ahead! A brief informational graphic of the history of parallel cluster processing and databases is provided in Figure 1.5.

1.5.3   CAP Theorem

The CAP theorem is a seminal theorem [13] used to specify what guarantees can be provided by a distributed database. The CAP theorem states that no distributed database can simultaneously provide strong guarantees on the consistency, availability and partition tolerance of a database. This is often stated as the two-out-of-three rule though in reality it is more of a loose guarantee rather than losing the guarantee completely. In practice partition tolerance is an important aspect of NoSQL distributed databases; the two-out-of-three rule of the CAP theorem implies that most such databases fall into a consistency-partition tolerance (CP) or availability-partition tolerance (AP) style. Traditional relational databases are examples of choosing consistency and availability (CA) as the two out of three CAP theorem guarantees.

Unlike the definition of consistency in ACID (which refers to a single node view of data in a database), consistency in the CAP theorem refers to the property that all nodes in a distributed database see the same data and provide a client with the same results regardless of which server is
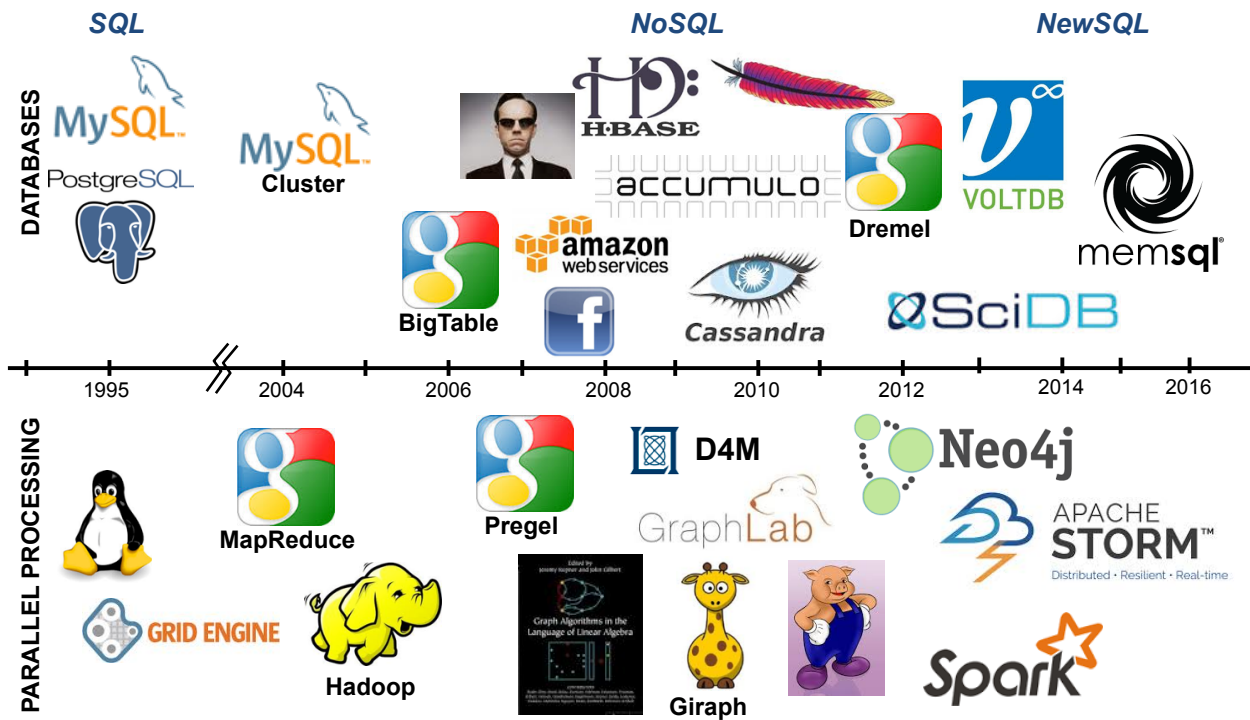
Figure 1.5: An incomplete history of open source databases and parallel cluster computing technologies.

responding to a client request. Very often, a strong consistency guarantee is enforced by placing locks on a table, column, row or cell until all parts of the transaction are performed. While this property is very useful to ensure that all queries subsequent to the completion of the transaction see the same value, locking can hinder performance and availability guarantees. For example, in the case of a partition, enforcing consistency implies that certain nodes will not be able to respond to requests until all nodes have a consistent view of the data thus compromising the availability of these nodes. A NoSQL database that prioritizes consistency over availability is Google BigTable [19] (though it may still have relaxed consistency between data replicas spread across database instances).

Database availability is a property in a distributed database which implies that every transaction must receive a response about whether the transaction has succeeded or failed. Databases that provide strong availabilty guarantees typically prioritize nodes responding to requests over maintaining a consistent system-wide view of the data. This availability, however, often comes at the

cost of consistency. For example, in the event of a database partition, in order to maintain availability, certain parts of a distributed database may provide different results until a synchronization occurs. An example of a NoSQL database that provides a strong availability guarantee is Amazon Dynamo [23] which provides a consistency model often called *eventual consistency* [66].

Consider the example transaction given in Figure 1.6. In this example, the transaction is to update the count of the word *Apple* in *Doc1* to be 5 from an application that computes a word count in documents. In a relational database, this transaction can occur within a single transaction that locks the row (*Doc1*) and performs the update before relinquishing the lock. In a highly available distributed database the supports eventual consistency, this update may be performed in parallel and eventually combined to show the correct count of the word Apple in *Doc1* to be 5. For a short period of time, until this consistency is achieved, different nodes may provide a different response when queried about the count of the word Apple in *Doc1*.
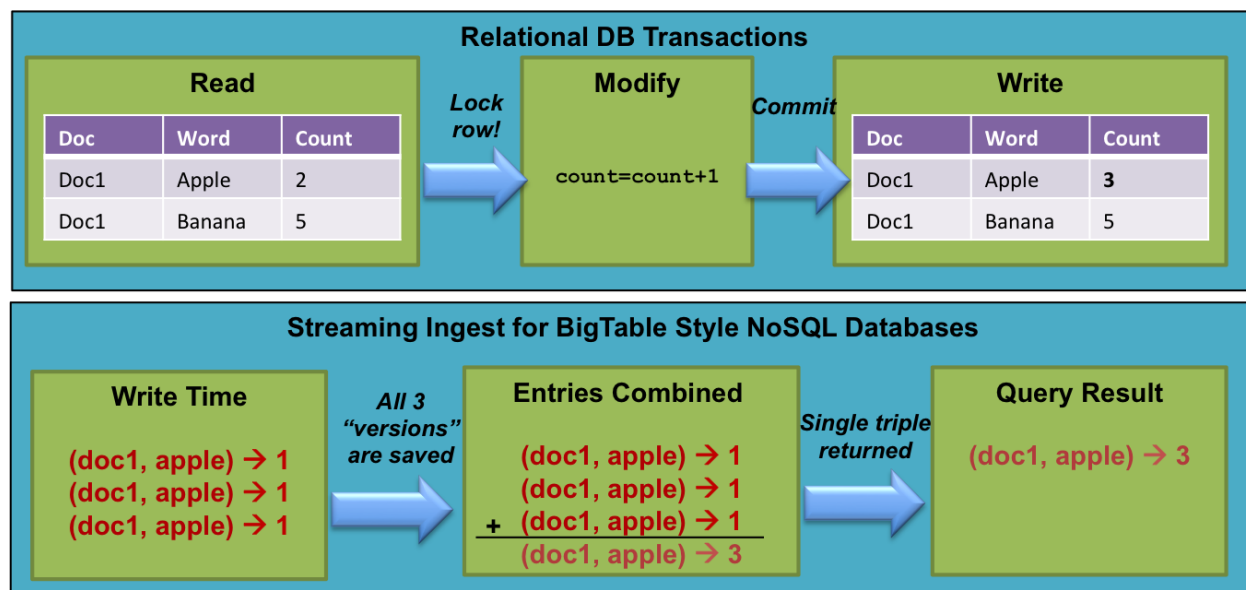
Figure 1.6: Relational update transaction compared with non-relational database update transaction.

The final aspect of the CAP theorem is database partition tolerance. Database partition tolerance is a database property that allows a database to function even after system failures. This property is often a fundamental requirement for large scale distributed databases. In the event of failure of a piece of a distributed database, a well designed database will handle the failure and move pieces of data to working components. This property is usually guaranteed by most of NoSQL databases.

Figure 1.7: Notional guide to implication of the CAP theorem for database design. Traditional relational databases provid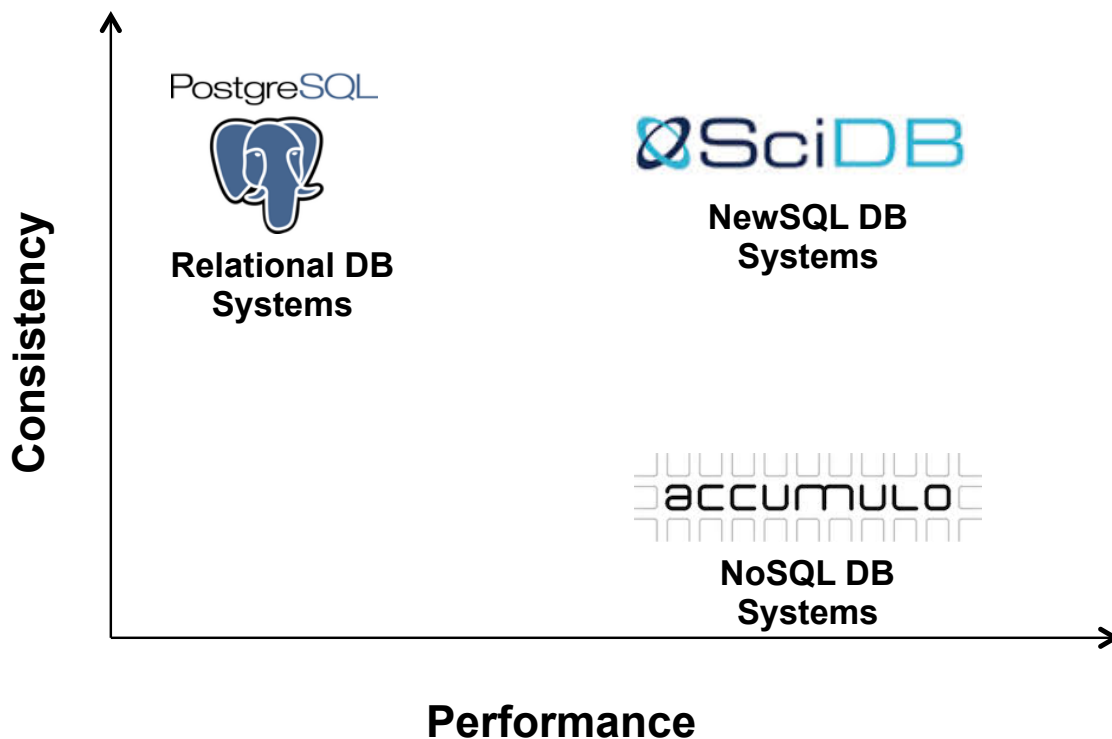e high consistency, NoSQL databases provide high performance at the cost of consistency, and NewSQL databases attemp to bridge the gap.

Traditional relational databases do not rely on distributed networks which are prone to disruption, thus avoiding the need for partition tolerance.

In recent years, there has been some controversy [57, 14, 34] surrounding the use of the CAP theorem as a fundamental rule in the design of modern databases. Most often, the CAP theorem is used to imply that one can have an all or nothing of two of the three aspects. However, it has been shown in [15] that careful partition and availability optimization may be able to achieve a database that provides a version of all three guarantees. While the CAP theorem can be used for high-level understanding of tradeoffs and design of current technologies, it is certainly possible to design databases that provide versions of guarantees on all three properties through different data models or hardware. In Figure 1.7, we provide a notional guide to the CAP theorem and database classes and also shows an example technology for each of these database classes.

1.5.4   Relational Databases

Relational databases such as MySQL, PostgreSQL, and Oracle form the bedrock of database technologies today. They are by far the most widely used and accessed databases. We interact with these databases daily: everywhere financial transactions, medical records, and purchases are made. From the CAP theorem, relational databases provide strong consistency and availability; however, they do not support partition tolerance. In order to avoid issues with partition tolerance in distributed databases, relational databases are often vertically scalable. Vertical scalability refers to systems that scale by improving existing software or hardware. For example, vertically scaling a relational database involved improving the resources of a single node (more memory, faster processor, faster disk drive, etc.). Thus, relational databases often run on high-end, expensive nodes and are often limited by the resources of a single node. This is in contrast to non-relational database that are designed to support horizontal scalability. Scaling a database horizontally involves adding more nodes to the system. Most often, these nodes can be inexpensive commercial off-the-shelf systems (COTS) that are easy to add as resource requirements change.

Relational databases provide ACID guarantees and are used extensively in practice. Relational databases are called *relational* because of the underlying data model. A relational database is a collection of tables that are connected to each other via relations expressed as keys. The specification of tables and relations in a database is referred to as the schema. Schema design requires thorough knowledge of the dataset. Consider a very simple example of a relational database that maintains a record of purchases made by customers as depicted in Figure 1.8. The main purchase table can be used to track purchases. This table is related to a customer table via the customer ID key. The purchase table is also connected to a product table via the product ID key. Using a combination of these tables, one can query the database for information such as "who purchased a banana on March 22, 2010?". Many databases may contain tens to hundreds of tables and require careful thought during the design.

Purchase Table

Transaction ID	Customer ID	Product ID	Purchase Date
1112	24221	8977	03-22-2010
1113	24222	8978	03-22-2010
1114	24223	8979	03-22-2010

Customer ID	Customer	Address
24221	Bob	123 East Street
24222	Alice	223 Main Street
24223	Martha	465 North Street

Customer Table

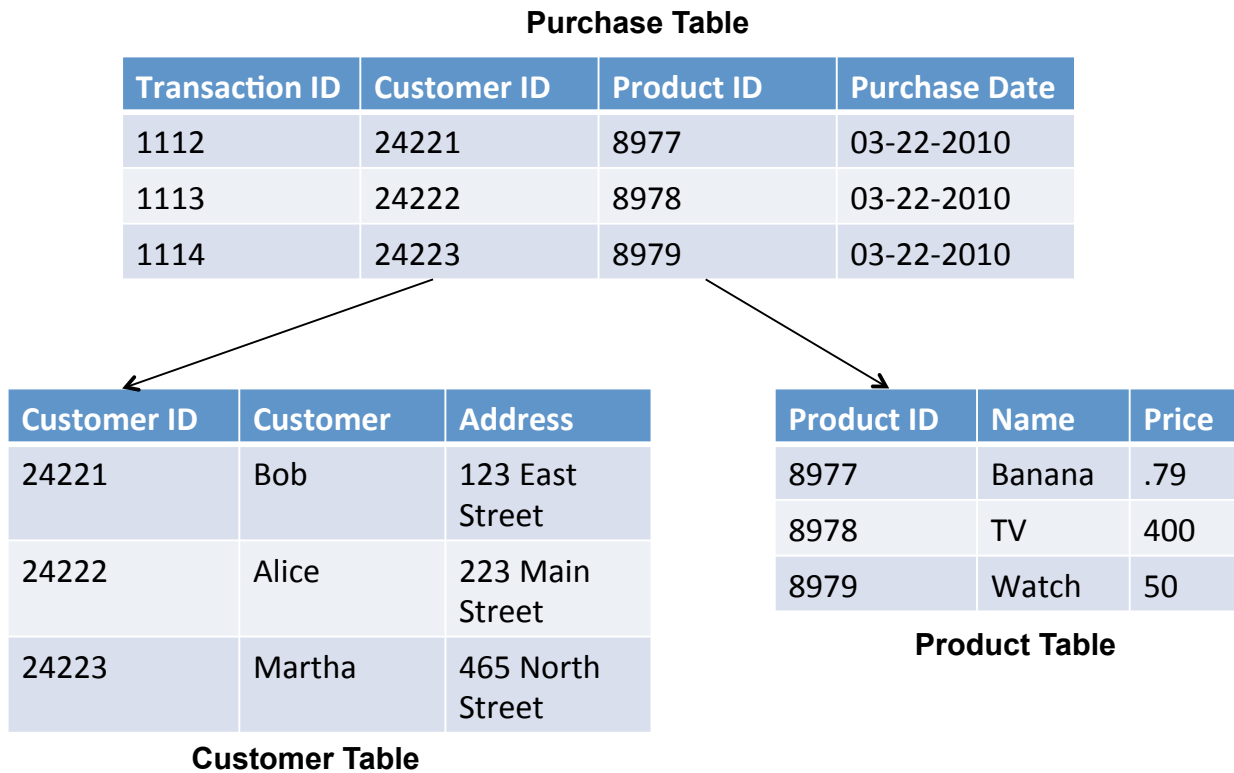| Product ID	Name	Price
8977	Banana	.79
8978	TV	400
8979	Watch	50

Product Table

Figure 1.8: A simple relational database that contains information about purchases made. The database consists of three tables: a purchase table, a customer table, and a product table.

1.5.5 NoSQL Databases

Since the mid-2000s and the Google BigTable paper, there has been a rise in popularity of Not Only SQL (NoSQL) databases. NoSQL databases support many of the large scale computing activities with which we interact regularly such as web searches, document indexing, large-scale machine learning, and graph algorithms. NoSQL databases support horizontal scaling: you can increase the performance through the addition of nodes. This allows for scaling through the addition of inexpensive commercial off-the-shelf systems as opposed to expensive hardware upgrades required for vertical scaling. NoSQL databases often need to relax some of the consistency or availability guarantees of relational databases in order to take advantage of strong partition tolerance guarantees. In order to keep up with rising data volumes, organizations such as Google looked for ways to incorporate inexpensive off-the-shelf systems for scaling their hardware. However, incorporating such systems requires the use of networks which can be unreliable. Thus, partition tolerance to network disruptions became an important design criteria. In keeping with the CAP theorem, either

consistency or availability must be relaxed to provide partition tolerance in a distributed database.

At a transaction level, NoSQL databases provide BASE guarantees. These guarantees may not be suitable for many applications where strong consistency or availability is required. However, for a variety of *Big Data* applications, BASE guarantees are sufficient for the purpose. For example, recall the example transaction described in Figure 1.6. If the end analytic (Step 5 in the Big Data pipeline) is an approximate algorithm to look for trends in word count, the exact count of the word Apple in *Doc1* may not be as important as the fact that the word Apple exists in the document. In this case, BASE guarantees may be sufficient for the application. Of course, before choosing a technology to use for an application, it is important to be aware of all design constraints and the impact of technology choice on the final analytic requirements.

NoSQL database use a variety of data models and typically do not have a pre-defined schema. This allows developers the flexibility to specify a schema that leverages the database capabilities while supporting the desired analytics. Further, the lack of a well defined schema allows dynamic schemas that can be modified as data properties change. Certain graph databases [8] use data structures based on graphs. In such databases, an implicit schema is often generated based on the graph representation of the data in the database. Key-value databases [35] take a given dataset and organize them as a list of keys and values. Document stores such as those described in [6] may use a schema based on a JSON or XML representation of a dataset. Figure 1.9 shows an example of a JSON based schema applied to the dataset shown in Figure 1.8.

1.5.6   New Relational Databases

The most recent trend in database design is often referred to as NewSQL databases. Given the controversy surrounding the CAP theorem, such databases attempt to provide a version of all three distributed database properties. These databases were created to approach the performance of NoSQL databases while providing the ACID transaction guarantees of traditional relational databases [58]. In order to provide this combination, NewSQL databases often employ different hardware or data models than traditional database management systems. NewSQL databases may also make use of careful optimizations on partitioning and availability in order to provide a version

```
{
Purchase: 1
customer: Bob
Product: Banana
Purchase Date: 03-22-2010
Purchase Price: 0.79
}

{
Purchase: 2
Customer: Alice
Product: TV
Purchase Date: 03-22-2010
Purchase Price: 400
}

{
Purchase: 3
...
```
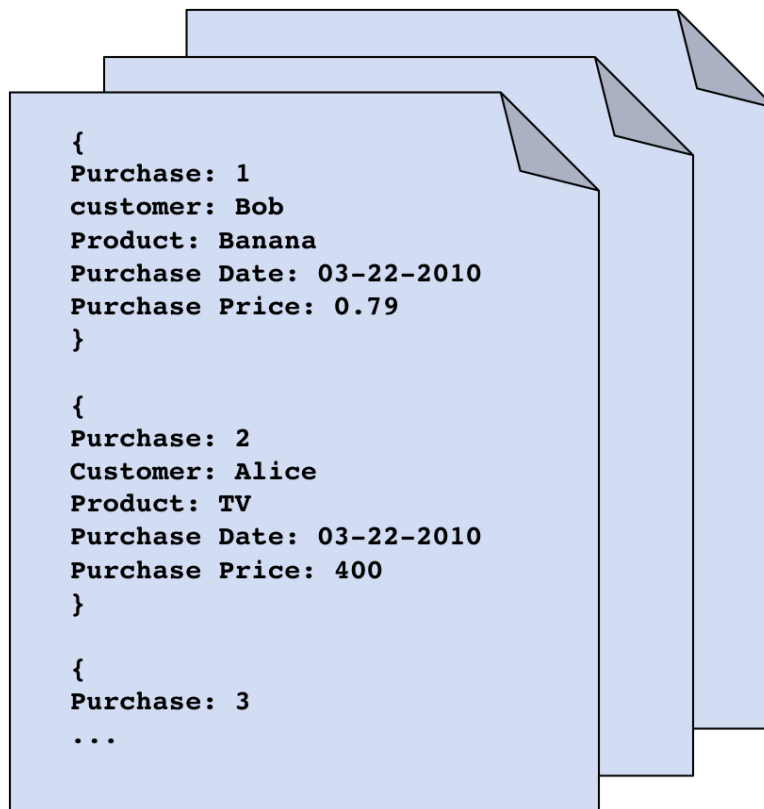
Figure 1.9: An example of using a JSON based data model for the purchase table of Figure 1.8.

of all three aspects of the CAP theorem. NewSQL databases may be considered as an alternative to both SQL and NoSQL style databases [43]. Most NewSQL databases provide support for the Structured Query Language (SQL).

NewSQL databases, while showing great promise, are a relatively new technology area. In the market now are databases designed for sensor processing [25], high speed online transaction processing (OLTP) [56], and streaming data and analytics [18].

A quick guide to the major differences between SQL, NoSQL and NewSQL style database is provided in Figure 1.10. Later in this section, we will provide a deeper look at a relatively new array based NewSQL database called SciDB.

	Relational Databases	NoSQL	NewSQL
Examples	MySQL, PostgreSQL, Oracle	HBase, Cassandra, Accumulo	SciDB, VoltDB, MemSQL
Schema	Typed columns with relational keys	Schema-less	Strongly-typed structure of attributes
Architecture	Single-node or sharded	Distributed, scalable	Distributed, scalable
Guarantees	ACID transactions	Eventually consistent	ACID transactions (most)
Access	SQL, indexing, joins, and query planning	Low-level API (scans and filtering)	Custom API, JDBC, Bindings to popular languages

Figure 1.10: A simple guide to differentiate between SQL, noSQL and newSQL style databases.

1.5.7   Deep Dive into NoSQL Technology

Apache Accumulo is an open source database used for high performance ingest and retrieval [54]. Accumulo is based on the Google Big Table. Apache Accumulo is a suitable technology for environments where large quantities of text data need to be indexed and inserted into a database at a high rate. In this deep dive, we will discuss the Accumulo data model, design and performance.

Data Model

Accumulo is a tabular key-value store where each cell or entry in Accumulo is a key (or tuple) mapped to a value. For each string labeled row and column, there is a unique value. Accumulo is a row store database, which means that look-up of a row by its row ID, can occur quickly. This property is very beneficial for large databases where the amount of data stored does not signifi-cantly increase the lookup time. By creating a suitable schema for a given dataset, this model can

Sparse Tuples:

(doc1, apple) → 1
(doc1, banana) → 5
(doc1, carrot) → 2
(doc2, banana) → 1
(doc2, daikon) → 2
(doc3, carrot) → 4
(doc3, eggplant) → 1

Logical Table:

Row ID	apple	banana	carrot	daikon	eggplant
doc1	1	5	2		
doc2		1		2	
doc3			4		1

Figure 1.11: The logical table (right) is stored as a series of tuples (left) in a key-value store database such as Accumulo.

be interpreted as semantic triples (subject, predicate, object), documents (mapping of text documents to values), large sparse tables or incidence matrices for graphs. One widely adopted Apache Accumulo schema is presented in [39]. In this schema, the original dense dataset it converted to a sparse representation through a four table schema. Two of the tables are used to encapsulate the semantic information contained in the dense dataset. A degree table is used to represent a count of entities which is useful for query planning and an additional raw table is used to store the original dataset in its full form.

To visualize what data stored in Accumulo may look like, consider Figure 1.11, which is a schema for a set of documents containing the names of fruits and vegetables. From a logical perspective, we can visualize a big table to look much like a very large spreadsheet in which many of the values are null or non-existent. However, similar to how sparse matrices are stored [55], Accumulo will only store non-zero entries in a manner similar to what is shown in the sparse tuples part of the figure. The row key (or row ID) refers to the document name or number, and the column key represents the name of the fruit or vegetable. The value in such a situation may represent the count or the number of times a particular fruit or vegetable was mentioned in a document. For example, in the figure, the fruit banana was mentioned in *doc*1 five times and one time in *doc*2.

In reality, the Accumulo data model is more complicated. Instead of just a three tuple (row, column and value), each cell is actually a seven tuple where the column is broken into three parts, and there is an additional field for a timestamp as seen in Figure 1.12.

Key					Value
Row ID	**Column**			**Timestamp**	
	Family	**Qualifier**	**Visibility**		

Figure 1.12: An Accumulo entry consists of a 7-tuple. Each entry is organized lexicographically based on the row key.

The column family is often used to define classes within the dataset. The column qualifier is used to define the unique aspect of the tuple. The visibility field is used to specify cell-level visibility for controlling who can access this data item. The value field is often used to store aspects of the dataset that must be stored but not searched upon. Given the Accumulo properties of fast row lookup, a good schema will usually store the semantic information of the dataset into the rows and columns of the table.

Design

Each key-value combination or cell in Accumulo is stored lexicographically and a number of cells are stored in tables. Tables can be arbitrarily large. Contrary to the design of relational tables, very often an entire dataset is put into a single table. In order to provide high performance distributed access, portions of a table are persisted on disk as tablets which are in turn stored in tablet servers as shown in Figure 1.13.

Since a single table can often be very large in a distributed database, each table is split into a number of tablets. A table in Accumulo is defined to be a map of key-value pairs (or cells) with a global order among the keys. A tablet is a row range within a table that is stored on a logical or physical computational element such as a server. A tablet server is the mechanism that hosts tablets and provides functionality such as resource management, scheduling, and hosting remote procedure calls.
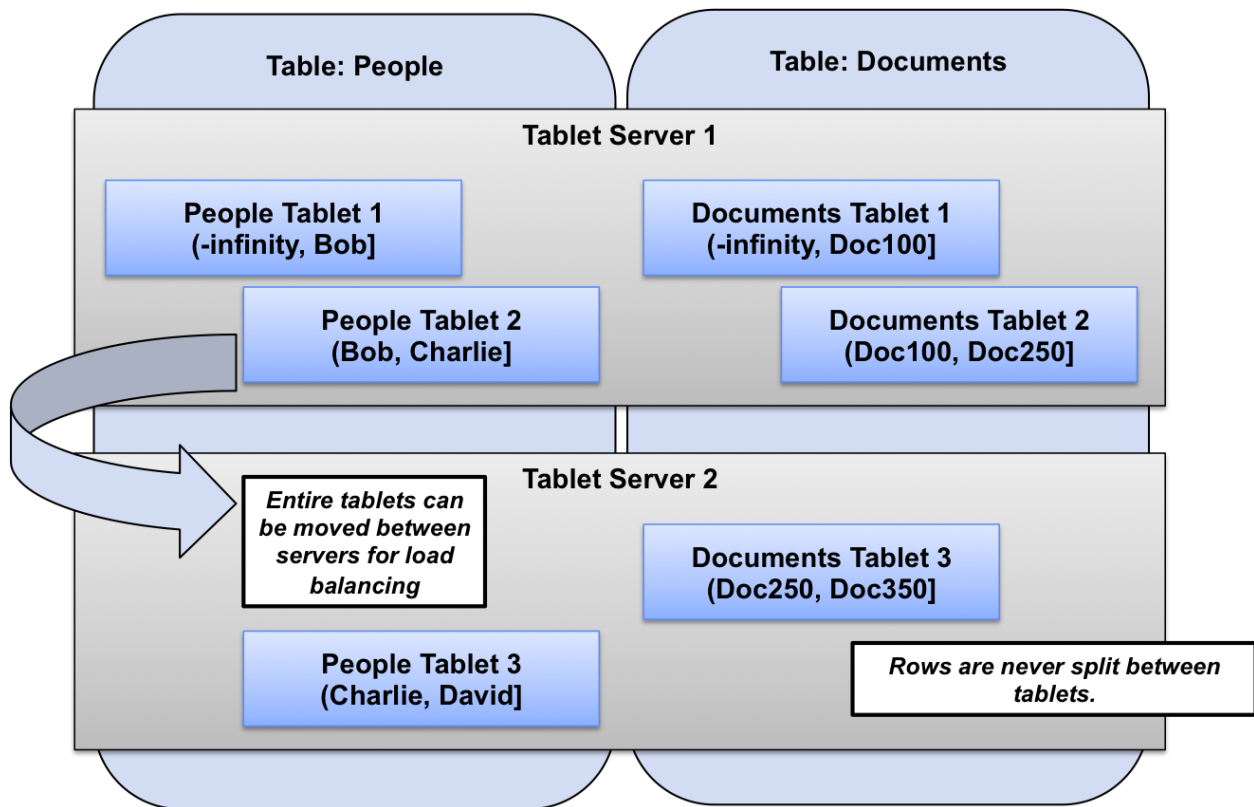
Figure 1.13: Accumulo tables are split into tablet which are hosted on tablet servers. Tablets associated with different tables may exist on the same tablet servers.

Performance

Accumulo supports high performance data ingest and queries. A test performed at MIT Lincoln Laboratory was able to demonstrate a database insert rate of nearly 115 million entries (key-value tuples) per second on an Apache Accumulo instance running on over 200 servers simultaneously [41]. The results from this experiment are shown in Figure 1.14.

1.5.8 Deep Dive into NewSQL Technology

NewSQL databases provide a new view into the design of databases. By shunning the popular CAP theorem, this style of database technology has the ability to combine the advantages of both SQL and NoSQL database for high-performance databases that provide ACID guarantees of single node relational databases. One technology in this category is an array-based database called SciDB. This
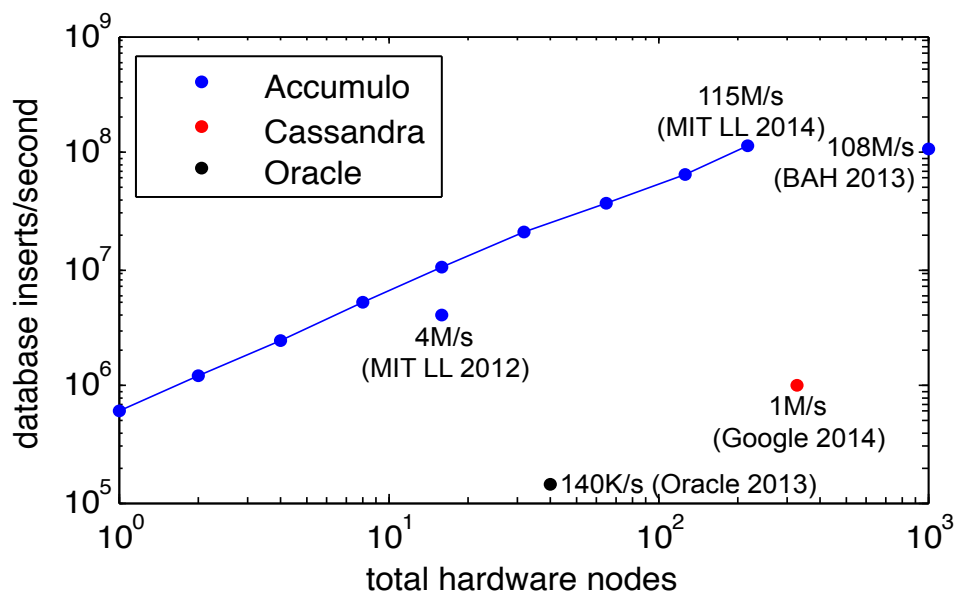
Figure 1.14: Demonstration of high performance data ingest in Accumulo. The y-axis is a measure of the number of 7-tuples shown in Figure 1.12 that Accumulo can insert per second.

database is designed for datasets and applications which can be represented as arrays or matrices. For example, SciDB is an ideal choice for signals or imagery which can be represented as a 1D or 2D array of values, respectively. SciDB is a massive parallel database with an array data model that enables complex analytics directly in the database. SciDB can be run on commodity or high performance computing clusters or in the cloud through services such as Amazon Web Services (AWS) [7]. A user can interact with SciDB through a series of SciDB connectors written in very high level programming languages such as R, Python, MATLAB, or Julia. In this deep dive, we will discuss the SciDB data model, database design and tested performance.

Data model

SciDB makes use of an array data model. Each cell of a SciDB array is a strongly typed structure of attributes. SciDB uses array indexing in which dimensions are essentially indices. Consider an example of using SciDB to store a topographic map. In a topographic map, the dimensions are latitude and longitude. As the value or attribute at a particular location, one usually stores the elevation or height of that location. An example of how this would look in SciDB is presented in Figure 1.15. The design of a SciDB schema is highly customized to an application.
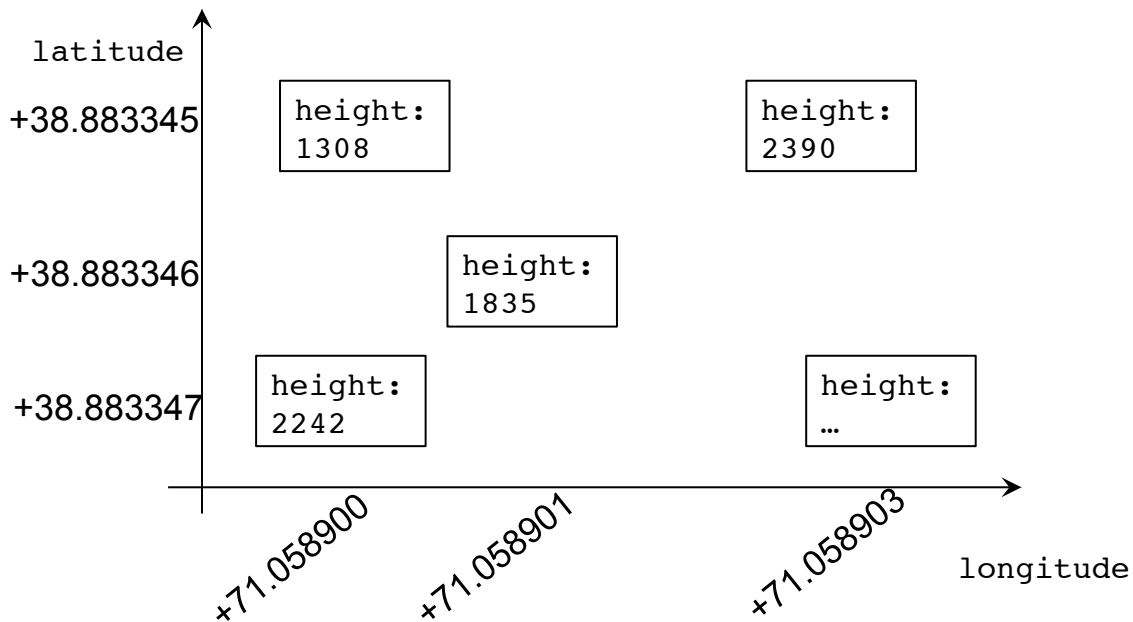
Figure 1.15: A notional example of a SciDB array for storing the height or elevation at a particular latitude and longitude.

SciDB also provides support for built-in analytics such those described in [60]. Using SciDBs built-in analytics for the dataset of Figure 1.15, one can find the latitude/longitude pairs which satisfy certain criteria such as an elevation threshold without moving the full dataset back to the client in order to perform the analysis.

Design

SciDB is deployed on a cluster of servers. Each server in the cluster has access to local processing, memory and storage capabilities. These servers can be in the cloud through services such as Amazon Elastic Compute Cloud (EC2) or hosted on high performance computing systems or even commodity clusters. Standard Ethernet connections are used to move data between servers. Each physical server hosts a SciDB instance (or instances) that is responsible for local storage and processing.

When a client or external application requests a connection with SciDB, it creates a connection with one of the instances running on a physical server in the cluster. A coordinator instance is

responsible for communicating query results to the connecting client and all other instances partic-
ipate in the execution of the query and data storage. The other instances in the cluster are referred
to as worker instances which work with the coordinator node.

Performance

SciDB benchmarking was performed at MIT Lincoln Laboratory. The data used for benchmarking
was generated using a random graph generator from the Graph500 benchmark [48]. The Graph500
scalable data generator can efficiently generate power-law graphs that represent common graphs
such as those generated by social media. The number of vertices and edges in the graph are set
using a positive integer called the SCALE parameter. Given a SCALE parameter, the number of
vertices, N, and the number of edges, M, are then computed as $N = 2^{SCALE}$ and $M = 8N$.

SciDB is a highly scalable database and is capable of connecting with multiple clients at once.
In order to test the scalability of SciDB, the test was performed using the parallel MATLAB tool,
pMATLAB [10], in addition to the Dynamic Distributed Dimensional Data Model (D4M) [40] to
insert data from multiple clients simultaneously. In order to overcome a SciDB bottleneck that
applies a table lock when data is being written to a particular table, D4M can create multiple tables
based on the total number of processes that are being used to ingest data. For example, if there are
four ingestors (four systems simultanously writing data to SciDB), D4M creates four tables into
which each ingestor will concurrently insert data. The resulting tables can then be merged after the
ingest using D4M if desired.

For this benchmark, SciDB was launched using the MIT SuperCloud [51] architecture through
the a database hosting system that launches a SciDB cluster in a high performance computing
environment. For the purpose of benchmarking SciDB on a single node, instances were launched
on a system with dual Intel Xeon E5 processors with 16 cores each and 64GB of RAM. For the
reported results, the SciDB coordinator and worker nodes were located on the same physical node.

The performance of SciDB is described using weak scaling. Weak scaling is a measure of the
time taken for a single processing element to solve a specific problem and measures the perfor-
mance when scaling with a fixed problem size per processor. In Figure 1.16, the performance

of SciDB for a SCALE that varies with the number of processors into SciDB is presented. The maximum performance (insert rate) was observed at 10 processors.
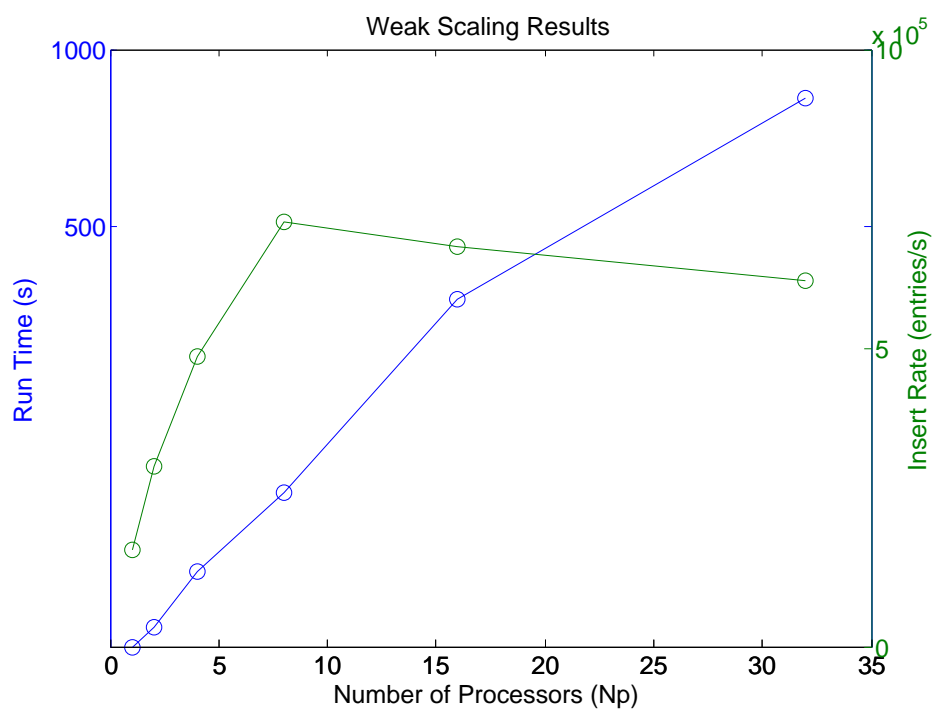

Figure 1.16: Weak scaling of D4M-SciDB insert performance for problem size that varies with number of processors.
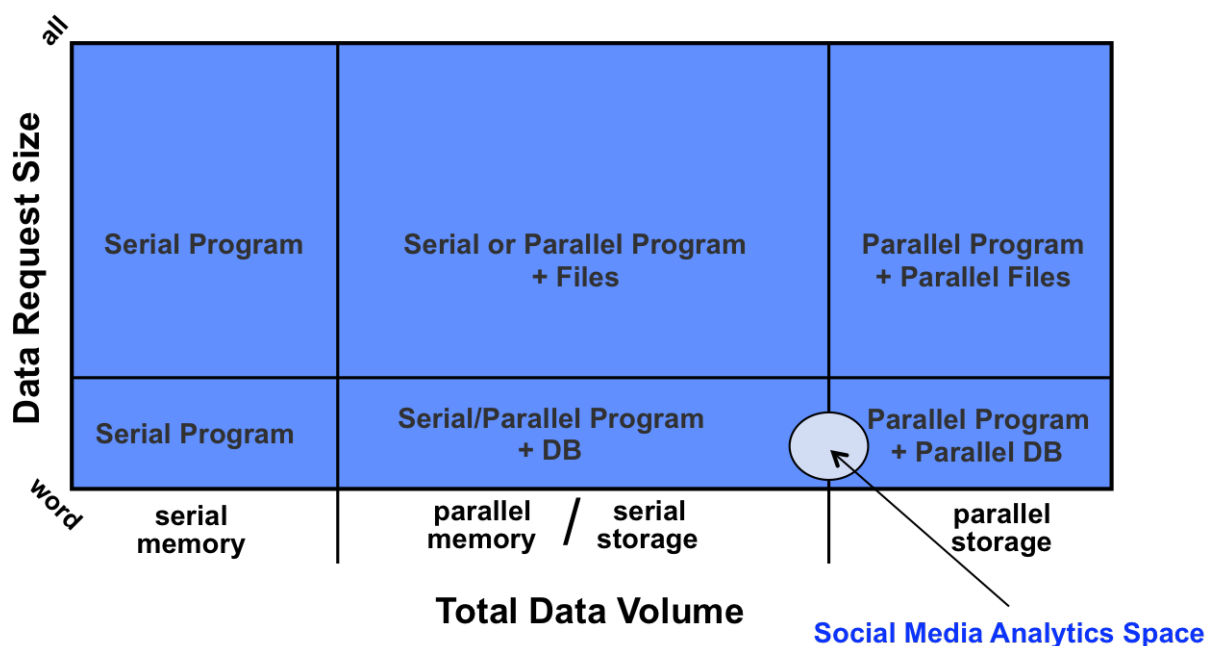
Figure 1.17: Choosing the right combination of technologies is a factor of many parameters such as total data volume and data request size. An example decision point for common social media analytics is shown in the light blue circle.

1.6 How to Choose The Right Technology

One of the guiding principles in technology selection is that "one size does not fit all" as mentioned in [61]. In the data storage world, this can translate to "there is no single technology that will solve all of your problems". For a given problem mapped to the pipeline of Figure 1.2, a typical solution may have a combination of filesystems, SQL, NoSQL and NewSQL databases and different parallel processing strategies. In fact, a good big data solution will make use of technologies that best satisfy the overall application goal. Choosing the right storage or database engine can be a challenging problem and often requires careful thought and a deep understanding of the problem being solved. Sometimes, one may go in looking for a database solution only to find that a database is not the right option! In Figure 1.17, we describe how various technologies can be selected based on the property of data size and data request size. Of course, this is a simple guide to technology selection and there may be many other factors that come into consideration when making a decision in the real world.

As described in Figure 1.17, when choosing a storage or database engine, one must take into account the data request size (as a percentage of the entire corpus), and the total data volume. For a small amount of data volume (10s of GB in 2015) and any request size, the most efficient solution is to make use of a systems onboard memory for the storage and retrieval of information. For a larger dataset (100s of GB to 10 TB in 2015), the most efficient solution for small requests (less than 5% of the entire dataset) is to use a database. If your request size is larger than approximately 5% of the entire dataset, it is often faster and more efficient to write a parallel program that operates on data in the file system. For much larger datasets (10TB to many PBs in 2015), one will need to make use of parallel storage technologies such as HDFS or Lustre or parallel databases such as Accumulo or SciDB. Again, if the total data request size is greater than 5% of the entire dataset, it may be more efficient to work directly on the files.

Databases are designed to pull small chunks of information out (finding a needle in a haystack) and not for sequential access (the forte of distributed filesystems). Much of the overhead incurred in using a database is for extensive indexing. Very often, this may be dictated by the application at hand, other components in a pipeline, and/or the desired interface and language support. For example, many relational databases support the Java Database Connector (JDBC), which is a convenient Java API for accessing relational databases. Performance characteristics and data properties may also dictate database choice. For example, for a rapidly changing data schema, key-value store NoSQL databases may be a suitable choice. For applications that require high performance and ACID compliance, NewSQL databases are a suitable option. The important thing to keep in mind is that technologies continue to evolve, and it is important to re-evaluate technology choices as requirements change or are not being satisfied.

1.7 Case Study of DBMSs with Medical Big Data

Medical big data is a common example used to justify the adage that "one size does not fit all" for database and storage engines. Consider the popular MIMIC II dataset [52]. This dataset consists of data collected from a variety of Intensive Care Units (ICU) at the Beth Israel Deaconess Hospital in Boston. The data contained in the MIMIC II dataset was collected over seven years and con-

tains data from a variety of clinical and waveform sources. The clinical dataset contains the data collected from tens of thousands of individuals and consists of information such as patient demographics, medications, interventions, and text-based doctor or nurse notes. The waveform dataset contains thousands of time series physiological signal recordings such as ECG signals, arterial blood pressure, and other measurements of patient vital signs. In order to support data extraction from these different datasets, one option would be to attempt to organize all the information into a single storage or database engine. However, existing technologies would prove to be cumbersome or inefficient for such a task. A distributed file system would provide inefficient random access to data (for example, looking up details of a patient). Similarly, a relational database would be inefficient for searching and operating on the waveform signals.

The next solution is to store and index each of the individual components into a storage or database engine that is the most efficient for a given data modality. In such a system, individual components of the entire dataset could be stored in the storage or database engine that best supports the types of queries and analytics one wishes to perform. In this system, one may place the clinical dataset in a relational database such as MySQL. The text notes may go into a NoSQL database such as Apache Accumulo and the waveform data may go into an array based NewSQL database such as SciDB. At MIT Lincoln Laboratory, we developed a prototype of such a system using these technologies [29].

The prototype developed supports cross-database analytics such as: "tell me about what happens to heart rate variance of patients who have taken a particular medication." Naturally, such a query needs information from the clinical data contained in MySQL database, the patient database contained in Accumulo and the waveform data contained in SciDB. The sample query provided is then broken up into three distinct queries where: 1) tell me which patients have taken a particular medication goes to MySQL, 2) tell me which of these patients have heart beat waveforms goes to Accumulo, and 3) show me what happened to these patients heart rate variance goes to the waveform database. At each of these sub-queries, associative arrays are generated that can be used to move the results of one query to the next database engine. In Figure 1.18, we show the web front end that uses the Dynamic Distributed Dimensional Data Model (D4M) to implement the query described above.
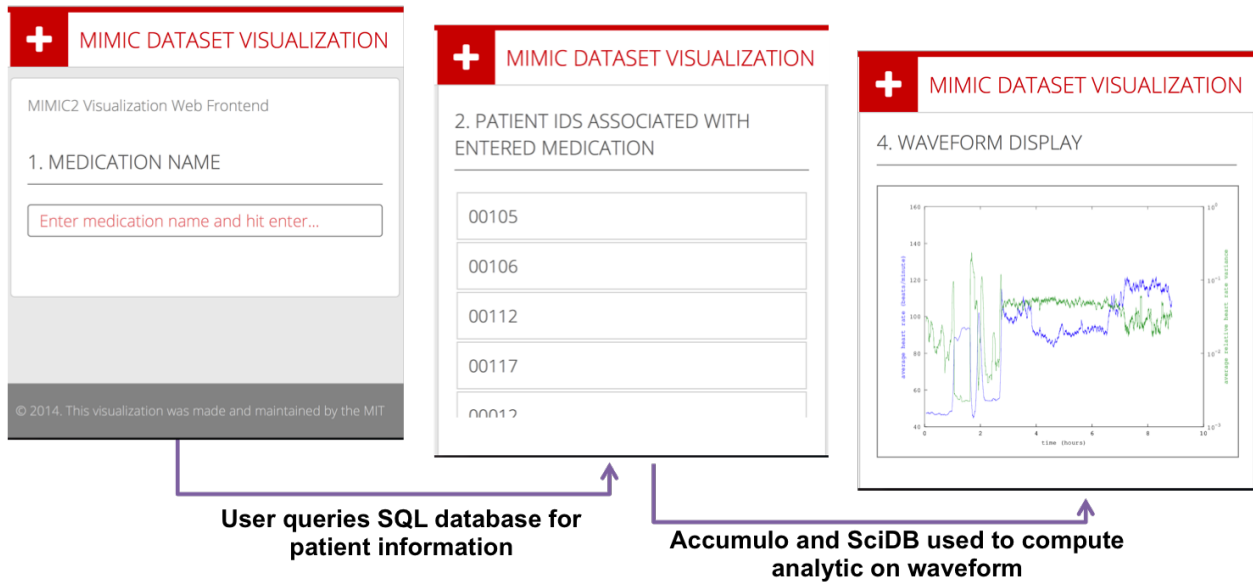
Figure 1.18: Screen shots of MIMIC II Visualization that uses a combination of SQL, NoSQL and NewSQL style databases in concert for a single analytic.

This example and many more highlight the importance of technology selection. By breaking up the problem into smaller pieces, choosing the right technology was much easier. Recall the pipeline of Figure 1.2. Filling in the details for steps 2 and 3 often requires careful consideration.

1.8 Conclusions

The world of storage and database engines is vast, and there are many competing technologies out there. The technologies we discussed in this chapter discuss methods to address three of the four V's of big data - volume (storage and database engines), velocity (distributed NoSQL and NewSQL databases) and variety (database schemas and data models). One V that we did not discuss is Big Data veracity - or privacy preserving technologies. For a thorough understanding of this topic, we recommend you read the chapter entitled: Cryptography for Big Data Security.

1.9   Acknowledgements

References

[1] Apache accumulo. *https://accumulo.apache.org/*.

[2] Es7k: World fastest entry-level lustre appliance. *DDN Product Brochure*.

[3] HDFS architecture guide. *http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html*.

[4] Mysql website. In *https://www.mysql.com/*.

[5] Oracle database 12c. *https://www.oracle.com/database/index.html*.

[6] Veronika Abramova and Jorge Bernardino. Nosql databases: Mongodb vs cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, pages 14–22. ACM, 2013.

[7] Amazon, Inc. Amazon ec2. *http://aws.amazon.com/ec2/*.

[8] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.

[9] Ching Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[10] Nadya Travinin Bliss and Jeremy Kepner. pMATLAB Parallel MATLAB Library. *International Journal of High Performance Computing Applications*, 21(3):336–359, 2007.

[11] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[12] Peter J Braam et al. The lustre storage architecture, 2004.

[13] Eric Brewer. A certain freedom: thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 335–335. ACM, 2010.

[14] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[15] Eric Brewer. Pushing the cap: Strategies for consistency and availability. *Computer*, 45(2):23–29, 2012.

[16] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[17] Daniel Carteau. Three interconnected raid disk controller data processing system architecture, December 11 2001. US Patent 6,330,642.

[18] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, et al. S-store: A streaming newsql system for big velocity applications. *Proceedings of the VLDB Endowment*, 7(13):1633–1636, 2014.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[20] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[21] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[24] Domo. https://www.domo.com/learn.

[25] Aaron Elmore, Jennie Duggan, Michael Stonebraker, Magda Balazinska, Ugur Cetintemel, Vijay Gadepally, Jeff Heer, Bill Howe, Jeremy Kepner, Tim Kraska, et al. A demonstration of the bigdawg polystore system. *Proceedings of the VLDB Endowment*, 8(12), 2015.

[26] Vijay Gadepally, Braden Hancock, Benjamin Kaiser, Jeremy Kepner, Pete Michaleas, Mayank Varia, and Arkady Yerukhimovich. Improving the veracity of homeland security big data through computing on masked data. In *IEEE Technologies for Homeland Security*, 2015.

[27] Vijay Gadepally and Jeremy Kepner. Big data dimensional analysis. *IEEE High Performance Extremem Computing Conference*, 2014.

[28] Vijay Gadepally and Jeremy Kepner. Using a power law distribution to describe big data. *IEEE High Performance Extremem Computing Conference*, 2015.

[29] Vijay Gadepally, Sherwin Wu, Jeremy Kepner, and Sam Madden. Mimicviz: Enabling visu-

alization of medical big data. In *New England Database Summit*, 2015.

[30] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows and biggest growth in the far east. *IDC iView*, 2012.

[31] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems (TODS)*, 8(2):186–213, 1983.

[32] Lars George. *HBase: the definitive guide*. O'Reilly Media, Inc., 2011.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[34] Seth Gilbert and Nancy Ann Lynch. Perspectives on the cap theorem. In *Computer 45, no. 2*. Institute of Electrical and Electronics Engineers, 2012.

[35] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.

[36] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, et al. Bluedbm: an appliance for big data analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 1–13. ACM, 2015.

[37] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[38] Jeremy Kepner. *Parallel MATLAB for multicore and multinode computers*, volume 21. SIAM, 2009.

[39] Jeremy Kepner, Christian Anderson, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Matthew Hubbell, Peter Michaleas, Julie Mullen, David O'Gwynn, et al. D4m 2.0 schema: A general purpose high performance schema for the accumulo database. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[40] Jeremy Kepner, William Arcand, William Bergeron, Nadya Bliss, Robert Bond, Chansup Byun, Gary Condon, Kenneth Gregson, Matthew Hubbell, Jonathan Kurz, et al. Dynamic distributed dimensional data model (d4m) database and computation system. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 5349–5352. IEEE, 2012.

[41] Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gade-pally, Matthew Hubbell, Peter Michaleas, Julie Mullen, Andrew Prout, et al. Achieving 100,000,000 database inserts per second using accumulo and d4m.

[42] Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gade-pally, Matthew Hubbell, Peter Michaleas, Julie Mullen, Andrew Prout, Albert Reuther, Antonio Rosa, and Charles Yee. Lustre, hadoop, accumulo. In *IEEE High Performance Extreme Computing*, 2015.

[43] Rakesh Kumar, Neha Gupta, Shilpi Charu, and Sunil Kumar Jangir. Manage big data through newsql. In *National Conference on Innovation in Wireless Communication and Networking Technology–2014, Association with THE INSTITUTION OF ENGINEERS (INDIA)*, 2014.

[44] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[45] Doug Laney. 3d data management: Controlling data volume, velocity and variety. *META Group Research Note*, 6, 2001.

[46] Yucheng Low, Joseph E Gonzalez, Aapo Kyrola, Danny Bickson, Carlos E Guestrin, and Joseph Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041*, 2014.

[47] Peter Mell and Tim Grance. The nist definition of cloud computing. *National Institute of Standards and Technology*, 53(6):50, 2009.

[48] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[49] Juan Piernas, Jarek Nieplocha, and Evan J Felix. Evaluation of active storage strategies for the lustre parallel file system. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 28. ACM, 2007.

[50] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[51] Albert Reuther, Jeremy Kepner, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Matthew Hubbell, Peter Michaleas, Julie Mullen, Andrew Prout, et al. Llsupercloud: Sharing hpc systems for diverse rapid prototyping. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[52] Mohammed Saeed, Mauricio Villarroel, Andrew T. Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H. Kyaw, Benjamin Moody, and Roger G. Mark. Mul-

tiparameter intelligent monitoring in intensive care ii (mimic-ii): A public-access intensive care unit database. *Critical Care Medicine*, 39:952–960, May 2011.

[53] Siddharth Samsi, Vijay Gadepally, and Ashok Krishnamurthy. Matlab for signal processing on multiprocessors and multicores. *Signal Processing Magazine, IEEE*, 27(2):40–49, 2010.

[54] Scott M Sawyer, B. David O'Gwynn, An Tran, and Tao Yu. Understanding query performance in accumulo. In *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, pages 1–6. IEEE, 2013.

[55] Rukhsana Shahnaz, Anila Usman, and Imran R Chughtai. Review of storage techniques for sparse matrices. In *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1–7. IEEE, 2005.

[56] Nikita Shamgunov. The memsql in-memory database system. In *IMDM@ VLDB*, 2014.

[57] Michael Stonebraker. Errors in database systems, eventual consistency, and the cap theorem. *Communications of the ACM, BLOG@ ACM*, 2010.

[58] Michael Stonebraker. Newsql: An alternative to nosql and old sql for new oltp apps. *Communications of the ACM*, 2012.

[59] Michael Stonebraker, Jacek Becla, David J DeWitt, Kian-Tat Lim, David Maier, Oliver Ratzesberger, and Stanley B Zdonik. Requirements for science data bases and scidb. In *CIDR*, volume 7, pages 173–184, 2009.

[60] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science & Engineering*, 15(3):54–62, 2013.

[61] Michael Stonebraker and Ugur Cetintemel. One size fits all: an idea whose time has come and gone. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 2–11. IEEE, 2005.

[62] Michael Stonebraker, Gerald Held, Eugene Wong, and Peter Kreps. The design and implementation of ingres. *ACM Transactions on Database Systems (TODS)*, 1(3):189–222, 1976.

[63] Michael Stonebraker and Lawrence A Rowe. *The design of Postgres*, volume 15. ACM, 1986.

[64] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[65] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, San-

jeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[66] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR*, volume 11, pages 134–143, 2011.

[67] Feiyi Wang, Sarp Oral, Galen Shipman, Oleg Drokin, Tom Wang, and Isaac Huang. Understanding lustre filesystem internals. *Technical Report for Oak Ridge National Laboratory*, 2009.

[68] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, Programming, and Applications: Software for Humanity*, pages 217–218. ACM, 2012.

[69] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, volume 10, page 10, 2010.